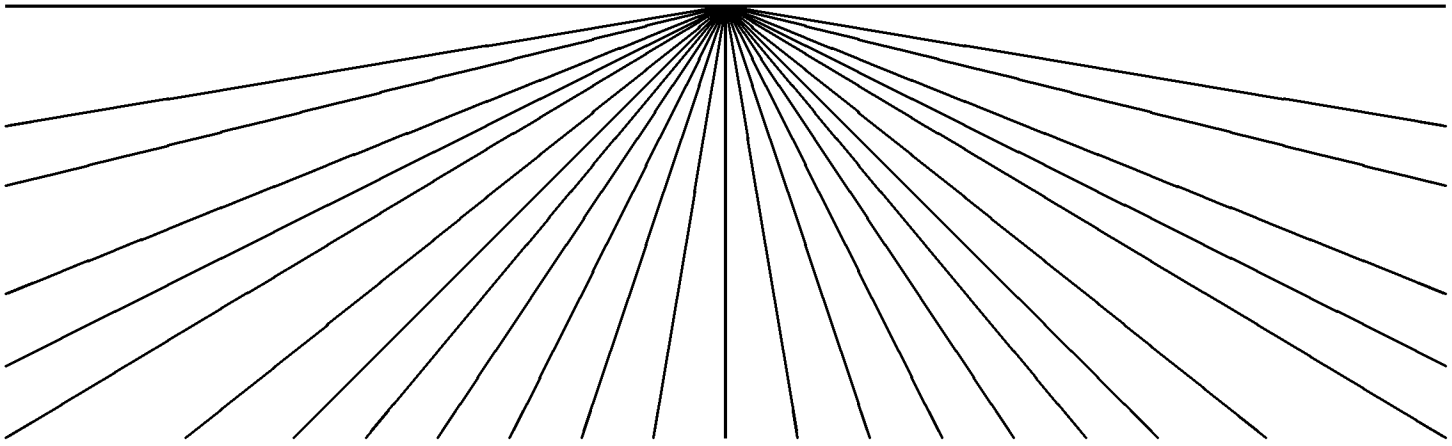
facultad de informática

universidad politécnica de madrid

A Technique for Dynamic Term Size Computation via Program Transformation

Manuel Hermenegildo
P. López García

TR Number CLIP 8/93.1(94)



A Technique for Dynamic Term Size Computation via Program Transformation

Technical Report Number: CLIP 8/93.1(94)
March 1994

Authors

Manuel Hermenegildo
herme@fi.upm.es
P. López García
pedro@dia.fi.upm.es

Departamento de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Keywords

Term Size Computation, Granularity Analysis, Parallelism.

Abstract

Knowing the size of the terms to which program variables are bound at run-time in logic programs is required in a class of applications related to program optimization such as, for example, granularity analysis and selection among different algorithms or control rules whose performance may be dependent on such size. Such size is difficult to even approximate at compile time and is thus generally computed at run-time by using (possibly predefined) predicates which traverse the terms involved. We propose a technique based on program transformation which has the potential of performing this computation much more efficiently. The technique is based on finding program procedures which are called before those in which knowledge regarding term sizes is needed and which traverse the terms whose size is to be determined, and transforming such procedures so that they compute term sizes "on the fly". We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We also discuss the advantages and applications of our technique and present some performance results.

Contents

1 Introduction 1

2 Overview of the Approach 2

3 Transforming Procedures: *Transformation Nodes* 4

4 Transforming Sets of Procedures: *Transformations* 8

5 Irreducible/Optimal Transformations 10

6 Searching for Irreducible Transformations 13

7 Experimental Results and Advantages of the Method 16

8 Conclusions and Future Work 19

9 Acknowledgements 19

References 20

1 Introduction

The need to know the size of the terms to which program variables are bound at run-time in logic programs arises in a class of applications related to program optimization which includes *granularity analysis* and selection among different algorithms or control rules whose performance may be dependent on such size. By term size we refer to measures such as list length, term depth, number of nodes in a term, etc.

For example, in granularity analysis the objective is to determine (or bound) a priori (i.e. before its execution) the number of steps that the execution of a given goal will involve. A number of researchers have investigated the automatic analysis of the (time) complexity of programs (see, for example, [5, 1, 6, 8, 11, 12, 13, 14]).

As pointed in [5], granularity analysis for a set of non recursive procedures is relatively straightforward. However, recursive procedures are somewhat more problematic: the amount of work done by a recursive call depends on the depth of recursion, which in turn depends on the size of the input. Reasonable estimates for the granularity of recursive predicates can thus be made only with some knowledge of the size of the input. In [5] a technique was presented for solving this problem based on performing a compile-time analysis which reduces granularity analysis work at run-time to evaluating simple functions of term sizes. However, the actual determination of those sizes in order to evaluate such functions is necessarily postponed until runtime.

The postponement of accurate term size computation to run-time appears inevitable in general since even sophisticated compile-time techniques such as abstract interpretation are based on computing approximations of variable substitutions for generic executions corresponding to general classes of inputs, while size is however clearly a quite specific characteristic of an input. Although the approximation approach can be useful in some cases we would like to tackle the more general case in which actual sizes have to be computed dynamically at run-time. Of course computing term sizes at run time is quite simple but at the same time it can involve a significant amount of overhead. This overhead includes both having to traverse significant parts of the term (often the entire term) and the counting process done during this traversal.

The objective of this paper is to propose a more efficient way of computing such sizes. The essential idea is based on the observation that terms are often already traversed by procedures which are called in the program before those in which knowledge regarding term sizes is needed, and thus that such sizes can often be computed "on the fly" by the former procedures after performing some transformations to them. While the counting overhead is not eliminated, overhead is reduced because additional traversals of terms are not needed. We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria.

The rest of the paper proceeds as follows: Section 2 presents an overview of the approach. Section 3 introduces our basic representations and Section 4 presents our concept of allowable transformations. Section 5 then introduces the concepts of irreducible and optimal transformations and highlights their important role. Section 6 then presents algorithms for finding

irreducible transformations and presents an example of the complete process. Section 7 shows some experimental results and finally, Section 8 presents our conclusions and suggestions for future work.

2 Overview of the Approach

As mentioned in the introduction, we are interested in transforming some predicates in such a way that they will compute some of their argument data sizes at run-time, in addition to performing their normal computation. It is often the case that because of previous transformations or other reasons, the size of certain terms is already known and it can be used as a starting point in the dynamic computation of those that we need to determine at a given point. Thus, we will be interested in the general problem of transforming programs to determine the sizes of one set of terms given that the sizes of the terms in another (disjoint) set are known.

Example 2.1 Consider, for example, the predicate `append/3`, defined as:

```
append([],L,L).
append([H|L],L1,[H|R])  :-  append(L,L1,R).
```

Suppose that we want to transform this predicate in order that it compute the length of its third argument. Observing the base case we can infer that the length of the term appearing in the third argument of the head is equal to that of appearing on the second argument after any successful computation. We can express this size relation as follows: $head[3] = head[2]$, where $head[i]$ denote the size of the term appearing at i^{th} argument position in the head. Thus, a transformation of this base case can be performed by adding two additional arguments, fourth and fifth, standing for the size of the term appearing in the second and third argument respectively:

```
append3i2([],L,L,S,S).
```

In this way, if we call the base case supplying the size of the second argument, we will obtain that of the third one once the call succeeds.

Observing the recursive clause, we can see that the size of the third argument of the head is equal to the size of the third argument of the first body literal plus one. We express this size relation as follows: $head[3] = body_1[3] + 1$, where $body_j[i]$ denotes the size of the term appearing at i^{th} argument position in the j^{th} literal of the body. Then we can think in using a transformed version of this body literal in order to compute $body_1[3]$. But to do this it is necessary that the size of the second argument of this body literal ($body_1[2]$) be supplied at the call (in order that $body_1[3]$ could be computed when recursion finishes. Since we have the following size relation: $body_1[2] = head[2]$, we can conclude that it is possible to compute the size of the third argument of append if the size of the second one is supplied at the call.

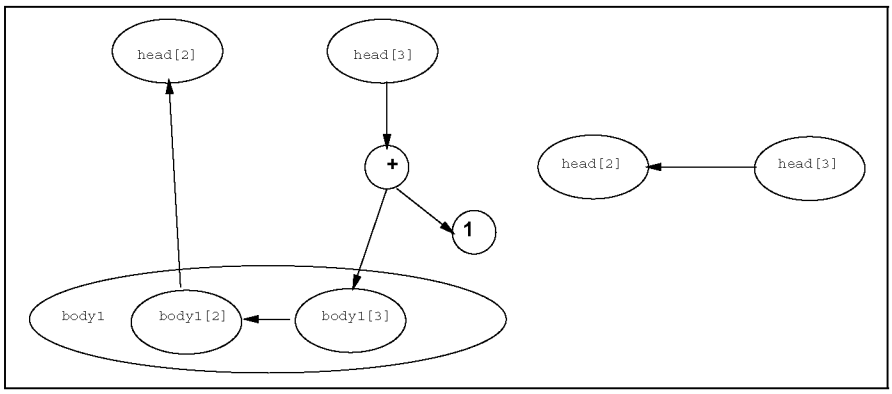


Figure 1: Size dependency graphs for predicate append/3.

The recursive clause can be trivially transformed as follows with the knowledge of the previous size relations: [1]

```
append3i2([],L,L,S,S).
append3i2([H|L],L1,[H|R],S2,S3)  :-  append3i2(L,L1,R,S2,Sb3), S3 is Sb3 + 1.
```

□

We can see that the problem can be reduced to finding what we can call a size dependency graph for each clause of the predicate to be transformed. Figure ?? shows the size dependency graphs corresponding to the previous example:

This graph is required to meet some conditions (which will be further explained in detail throughout the paper) in order that the derived predicate transformation correctly computes the desired term sizes. In other words, it is necessary to know for each clause, and for each term occurring at a head position whose size is going to be computed by the transformed procedure at run-time, an expression which gives the size of the term as a function of the sizes of other positions in the clause, which in turn can be computed by applying the same technique. Informally, the set of size dependency graphs constitute the information needed to transform a predicate, and is represented by means of what we call a *transformation node*. In general it is necessary to transform more than one predicate to perform a particular size computation. In this case, transformation nodes will constitute nodes in a search tree (the search algorithm is explained in detail in Section 6) tailored to finding a set of such nodes leading to a program transformation which correctly computes the desired term sizes.

[1] For clarity, this class of transformations is used in the examples even if they are not ideal given that they destroy tail recursion optimization. However it is quite straightforward to perform the equivalent transformation which preserves tail recursion optimization by using an accumulating parameter. These are the transformations performed in practice. Note also that although presenting the technique proposed in terms of a source-to-source transformations is useful both in the exposition and as a viable implementation technique, the transformation can also be implemented at a lower level in order to reduce the run-time overheads involved even further.

Roughly, our approach consists first in inferring all possible size relations between arguments of the program clauses which can be involved on the desired size computation. [2] In [5], a data dependency-based method for statically estimating these argument size relations is described. Secondly, all possible transformation nodes are constructed from these size relations, and finally, find the set of transformation nodes leading to correct size computations. While constructing a transformation node we make the assumption that any transformed version of any predicate (distinct of the transformed version corresponding to the transformation node itself) can be correctly carried out, and afterwards, this assumption will be checked by means of the search algorithm.

It should be noted that in some simple cases, similar transformations to the ones we propose can be obtained by adding to the original program some code that would perform the size computation in a naive way, and then partially evaluating that code into previous procedures. Note, however, that the algorithm presented herein is on one hand simpler than this approach, in the sense that it avoids having to include in the compiler a complete partial evaluator, which may not be otherwise needed. And, on the other hand, it is also more powerful (in this particular application) in that if several possible transformations are suitable, it constructs those which have the least runtime overhead, based on the criteria of choosing those which traverse less data and perform less arithmetic operations. Moreover, the search itself finds out dynamically which are the arguments whose sizes are strictly necessary to compute in order to serve as starting point for other size computations, something that would have to be given ahead of time to the partial evaluation approach. Moreover, we distinguish between "intra-literal" argument size relations, which refer to size relations between the argument positions of a single literal, and "inter-literal" argument size relations, which refer to relations between argument positions of different literals or the clause head. In this way, we can profit from some intra-literal argument size relations so that transformed literals need to traverse less data. This is so, because a size computation can then be performed directly in one operation, rather than by counting during the execution of the literal. Thus, at one program point we may decide to perform an arithmetic operation, provided that the needed sizes are known, or make a literal perform size computation by transforming it to perform data traversal.

3 Transforming Procedures: *Transformation Nodes*

In this section we explain how the information needed for procedure transformation is represented. We also formulate some conditions that this information has to meet in order for the transformation to lead to correct size computations. We thus prepare the way to end with the definition of a *transformation node*, which can be considered as a data structure which contains the information needed to transform a procedure (argument size relations etc.). Transformation nodes will also later be nodes in a search tree when the algorithm used to find different forms of transforming sets of procedures, or whole programs is presented.

Definition 1 (Label) *a structure, $lab(Pred, Os, Is)$, where:*

[2] We can consider only predicates in the strongly connected component of the call graph corresponding to the predicate which constitute the entry point of the transformation.

- *Pred: is the name and arity of the predicate to be transformed.*
- *Os: is a tuple of argument positions (represented as numbers) whose sizes are computed by the transformed predicate at run-time.*
- *Is: is a tuple of argument positions whose size are needed to compute the size of argument positions in Os. These sizes must be supplied at the predicate call (perhaps by previous computations).*

The condition: $Os \cap Is = \emptyset$ *is required.* □

With the above defined labels we can express which predicate *Pred* is transformed and which argument sizes will be computed as a function of which others. Transformation nodes will be labeled with such labels. An example of a label is: $lab(append/3,(3),(2))$,which states that the predicate $append/3$ will be transformed to compute the size of its third argument, provided that the size of the second one is supplied at the procedure call. This means that it is necessary to add two extra arguments to the transformed predicate which will stand for the sizes of the second and third argument of $append/3$.

Definition 2 (Size Descriptor) *a structure of the form:*

$$sd(lab(Pred, Os, Is), ArNum, LitNum, (Exp1, .., ExpN))$$

where:

- $lab(Pred, Os, Is)$ *is a label;*
- $LitNum$*: is a literal number in a clause (literals are numbered from left to right, starting by assigning one to the literal after the head);*
- $ArNum$*: is an argument number of literal* $LitNum$*; and,*
- $Exp1, .., ExpN$ *: are* Valid Size Expressions*, to be defined shortly.*

The condition: $ArNum \in Os$ *is required.* □

A size descriptor describes the size of a term appearing in a body clause. It supplies information about the position in the body (given by $LitNum$ and $ArNum$) at which the term occurs, what sizes are computed by the literal in which the term appears, and which are the terms whose size is needed for this computation. The condition $ArNum \in Os$ states that the size required has to be computed by the transformed literal. $lab(Pred, Os, Is)$ describes the size computation for which the literal $LitNum$ is transformed. $Exp1, .., ExpN$ describe the sizes of the terms that occur at arguments of the literal number $LitNum$ in Is. These sizes are needed for the transformed literal to perform the computation of terms appearing at argument positions in Os.

An example of size descriptor may be: $sd(lab(append/3,(3),(2)),3,1,(h(2)))$. This can be read as: "The size of the term appearing at argument number 3 of the body literal number 1.

The predicate of this literal is a transformation of the predicate *append*/3 in order to compute the size of its third argument, provided that the size of the second one is supplied at the call. Moreover, the size of this second argument is equal to the size of the second argument of the head ($h(2)$)".

Definition 3 (Size Expression) *A* size expression *is recursively defined as follows:*

- *A Natural number is a* size expression.
- *A term* $h(i)$*, is a* size expression.
- *A Size Descriptor is a* size expression.
- *If* E_1 *and* E_2 *are* size expressions*, then* $E_1 \triangle E_2$ *is a* size expression*, where* $\triangle$ *is any usual arithmetic operator (*$+$*,* $-$*,* $*$*, exponentiation, etc.).*

$h(i)$ *denotes the size of argument number* i*, or position number* i *of a clause head.* □

Definition 4 (Valid Size Relation) *A size relation is valid if it is true for every substitution that makes the terms occurring in such size relation ground.*

In the following, and for the sake of brevity, we will refer to valid size relations as size relations.

Definition 5 (Valid Size Expression) *a size expression* Exp *is valid if for each size descriptor:*

$$sd(lab(Pred, Os, Is), ArNum, LitNum, (Exp1, .., ExpN))$$

appearing in Exp*, and for each literal number* n *appearing in the size descriptors of* $(Exp1, .., ExpN)$*,* $n < LitNum$*.*

□

A valid size expression provides information about the size of some term in a clause. If the valid size expression is a head position ($h(i)$), then it represents the size of the i^{th} argument of the head.

Definition 5 establish that the sizes supplied to a transformed literal can be computed only by previous literals of the body. This requirement is due to the fact that the sizes supplied have to be "ground" at the call, because we are interested in using built-ins similar to "is/2" (in fact, more efficient and specialized versions) to perform the arithmetic operations needed to compute sizes and these built-ins require all but one of their arguments to be ground. It is important to note that this condition may be relaxed if the target language is for example a Constraint Logic Programming language [7] which can solve linear equations. However actual equation solving would probably incur in significant overhead. Thus we enforce the condition

both for efficiency reasons and for allowing the transformed programs to be executed without requiring any constraint solving capabilities in the target language.

An example of a valid size expression, taken from Example 2.1 is:

$$1 + sd(lab(append/3,(3),(2)),3,1,(h(2)))$$

which states that in the expression $1 + body_1[3]$, $body_1[3]$ can be computed by transformed literal number 1, provided that $body_1[2]$ is supplied, and that $body_1[2] = head[2]$, i.e., that it is in fact supplied by the head.

Once we have all necessary definitions we define the concept of transformation node.

Definition 6 (Transformation Node) *is a pair*

$$(lab(Pred, Os, Is), SizeAssignment),$$

where $lab(Pred, Os, Is)$ is a label which is the label of the node. $SizeAssignment$ is a tuple of n clause assignments*, n being the number of clauses in predicate $Pred$. Each such assignment refers to a different clause of $Pred$, and is a tuple of m items, where m is the cardinality of Os. There is an item for each argument number in Os. Each such item is a pair:*

$$(ArNum, VSE),$$

This pair describes the size of the term appearing at position number $ArNum$ of the clause head (denoted $head[ArNum]$) in relation with the sizes of other terms appearing at some clause positions. $ArNum$ is an argument number, $ArNum \in Os$, VSE is a valid size expression and:

1. *$head[ArNum] = SR(VSE)$ is a valid size relation. Where $SR(VSE)$ is a size relation obtained from VSE by replacing the size descriptors which does not appear in other size descriptor by the term $size(Term)$, $Term$ being the term occurring at the position indicated by the size descriptor, and replacing the size expressions of the form $h(i)$ by $head[i]$. E.g. $SR(1 + sd(lab(append/3,(3),(2)),3,1,(h(2)))) = 1 + size(R)$ (Note that $size(R) \equiv body_1[3]$), and $head[3] = 1 + size(R)$ is a valid size relation for the recursive clause of the predicate* `append`*/3.*
2. *Let I be the set of all argument numbers i such that $h(i)$ appears in some size expression of $SizeAssignment$, then $I = Is$.*
3. *If L is a label, $L = lab(Pred, Os, Is)$, appearing in some size descriptor $sd(L, ArNum, LitNum, Exp)$ of a clause assignment then for each $ArNum' \in Os$ exists a size descriptor in the same clause assignment of the form $sd(L, ArNum', LitNum, Exp')$. In other words, all the sizes of the terms appearing at positions indicated in Os are actually needed.*
4. *If*

$sd(LAB1, ArNum1, LitNum1, SizeExp1)$ *and*
$sd(LAB2, ArNum2, LitNum2, SizeExp2)$

are two size descriptors appearing in the same clause assignment, and $LitNum1 = LitNum2$*, then*

$LAB1 = LAB2$ *and* $SizeExp1 = SizeExp2$.□

Condition 2 states that all the term sizes that are needed from a clause head are actually supplied by it.

Condition 4 states that a body literal can only be transformed in one way, and that the sizes supplied to it can be computed also in only one way.

Example 3.1 Consider Example 2.1, where a procedure transformation is proposed for the predicate `append/3`. The information needed for this transformation can be represented with the following transformation node:

$(lab(append/3, (3), (2)),$
$(((3, h(2))),$
$((3, 1 + sd(lab(append/3, (3), (2)), 3, 1, (h(2)))))))$

The procedure transformation process is trivial given this information.

The first clause assignment $((3, h(2)))$ corresponds to the first clause of the predicate `append/3` (basic case), and the second one corresponds to the recursive clause. □

The intuition which can be gathered from the previous example is that it is possible to perform the size computation at run-time if some conditions hold on the transformation nodes. This will be the subject of the following sections.

4 Transforming Sets of Procedures: *Transformations*

In this section we deal with the problem of transforming sets of procedures which form part of a call-graph, in order that they perform a size computation. In this case it is necessary to have at least a transformation node for some of them and these nodes have to meet some conditions that are explained below. To define the concept of *Transformation*, which informally can be considered as the information needed to transform a set of procedures, we need the following definitions:

Definition 7 (*Con* relation) *We define a relation,* Con *between transformation nodes as follows:*

$(N_1, N_2) \in Con$ *if and only if the label of* N_2*,* LAB_2 *appears in some size descriptor of the size expressions of* N_1*, i.e. there is a size descriptor in* N_1 *of the form:*

$sd(LAB_2, ArNum, LitNum, (Exp1, .., ExpN))$ □

Definition 8 (Connected nodes) *Given a transformation node EP and a set of transformation nodes, TNS, we define the set of connected transformation nodes, $CN(EP, TNS)$ as:*

$$CN(EP, TNS) = \{N \in TNS | (EP, N) \in Con^T\},$$

where Con^T is the transitive closure of Con.□

Definition 9 (Ordering between labels) *Given two labels,*

$$X = lab(Pred, Os, Is_x) \text{ and } Y = lab(Pred, Os, Is_y),$$

we say that $X < Y$ if and only if $Is_x \subset Is_y$. □

For example:

$lab(append/3, (3), (2)) < lab(append/3, (3), (1, 2))$, but
$lab(append/3, (3), (2)) \not< lab(append/3, (3), (1))$

Definition 10 (Transformation) *A pair (TNS, EP), where EP is a transformation node, and TNS is a set of transformation nodes, is a* Transformation *if and only if:*

1. $EP \in TNS$.
2. *Let $NS = \{EP\} \cup CN(EP, TNS)$, then:*
   For each size descriptor:
   $$sd(lab(Pred, Os, Is), ArNum, LitNum, (Exp1, .., ExpN))$$
   appearing in the size expressions of the nodes in NS there is a transformation node in NS labeled with $lab(Pred, Os, Is)$.

EP is called the entry point *of the transformation.* □

Example 3.1 constitutes a transformation, where the entry point is the node itself.

Example 4.1 Consider the predicate qsort/2 defined as:

```
qsort([],[]).
qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        qsort(Ls,Ls2),qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

Let be N_1 the transformation node:

$$(lab(qsort/2,(2),()),\\ (((2,0)),\\ ((2,sd(lab(append/3,(3),(2)),3,4,\\ (1+sd(lab(qsort/2,(2),()),2,3,()))))))))$$

Let be N_2 the transformation node from Example 3.1, then, the pair $(\{N_1,N_2\},N_1)$ is a transformation, with entry point the node N_1 □

Definition 11 (Size Computation Specification) *We define a* size computation specification *as a pair $(Pred,Os)$, where $Pred$ is the name and arity of the predicate to be transformed, and Os is a tuple of argument numbers whose sizes are computed by the transformed predicate at run-time.* □

Definition 12 (Transformation for a size computation specification) *A Transformation for a size computation specification $(Pred,Os)$, is a transformation (T,EP) such that the label of EP is of the form $lab(Pred,Os,Is)$.* □

Theorem 1 *If there is a Transformation (T,EP), for a size computation specification $(Pred,Os)$, such that the label of EP is $lab(Pred,Os,Is)$, then it is possible to transform the clauses of $Pred$ to obtain a transformed Predicate $Pred'$, such that $Pred'$ computes the sizes of the arguments indicated in Os, provided that the sizes of arguments indicated in Is are supplied, besides of course performing the same computations that $Pred$ does.*□

Proof Trivial, by induction on the number of resolutions. □

Section 5 discusses how we can choose the best transformations.

A note on the generation and nature of transformation nodes: this generation is performed through a mode analysis to determine data flow patterns [3, 4, 9, 10, 2] and an argument size analysis [5]. It is important to note that this combined analysis can in some cases infer intra-literal size relations between arguments of a predicate. For example, it is possible to infer, for the predicate `append/3`, that the length of its third argument is the sum of its two first arguments, i.e. $h(3) = h(1) + h(2)$. This information can be used to generate transformation nodes which can form part of a transformation, but which need to traverse less data because a size computation can be performed directly in one operation, rather than by counting during the execution of the predicate. Thus, at one program point we may decide to perform an arithmetic operation, provided that the needed sizes are known, or make a literal perform size computation by transforming it to perform data traversal.

5 Irreducible/Optimal Transformations

Since there may be many possible transformations for a given size computation specification, we are interested in those involving the least amount of overhead at run-time. Such overhead is

dependent on the system, since it depends on the cost of argument passing and that of arithmetic operations. Reducing this overhead suggests considering transformations having the minimum number of transformation nodes and each of them having the minimum number of items in Is, where $lab(Pred, Os, Is)$ is the label of any node in the transformation. That is, to transform a predicate to make it compute the sizes of some of its arguments, we would like to know which are the arguments whose sizes are strictly necessary to perform this computation (in order to add only the absolutely necessary additional arguments and operations to the transformed predicates) and also what is the minimum number of predicates which have to be transformed. We first introduce the concept of *irreducible transformation* and show that to determine whether it is possible to transform a predicate, we only need to find irreducible transformations. Then we present some ideas regarding the generation of optimal irreducible transformations.

Definition 13 (Irreducible Transformation) *A transformation (T, EP), is* Irreducible *if and only if:*

1. *There is only one transformation node in T labeled with the same label.*
2. *$T = \{EP\} \cup CN(EP, T)$. That is, all the transformation nodes are needed.*
3. *There are no two transformation nodes in T, labeled with the labels X and Y respectively, such that $X < Y$.* □

The transformation shown in Example 4.1 is irreducible. We are going to show that to determine whether it is possible to transform a predicate, we only need to find irreducible transformations, but to do this we need some definitions and lemmas.

Definition 14 (Substitutions of labels: Sub(E,X,Y)) *Given a size expression E and two labels X and Y, where: $X = lab(Pred, Os, Is_x)$, $Y = lab(Pred, Os, Is_y)$ and $Is_y \subset Is_x$, we define $Sub(E, X, Y)$ as:*

1. *If $E = sd(Z, ArNum, LitNum, SizExp)$, where $SizExp = (E_1, ..., E_n)$ then:*
   (a) *If $Z = X$:*
   Let $Exp_1, ..., Exp_m$ be the size expressions in $SizExp$ associated with the argument numbers $a_1, ..., a_m \in Is_x$, such that $\{a_1, ..., a_m\} = Is_y$, then:
   $Sub(E, X, Y) = sd(Y, ArNum, LitNum, SizExp')$,
   where $SizExp' = (Sub(Exp_1, X, Y), ..., Sub(Exp_m, X, Y))$
   (b) *If $Z \neq X$ then :*
   $Sub(E, X, Y) = sd(Z, ArNum, LitNum, SizExp'')$,
   where $SizExp'' = (Sub(E_1, X, Y), ..., Sub(E_n, X, Y))$
2. *If $E = E_1 \triangle E_2$, where $\triangle$ is an arithmetic operator ($+$, $-$, $*$, etc.), then:*
   $Sub(E, X, Y) = Sub(E_1, X, Y) \triangle Sub(E_2, X, Y)$
3. *Otherwise: $Sub(E, X, Y) = E$.*

We can extend the previous definition to transformation nodes. Given a transformation node N and two labels X and Y, where:

$$X = lab(Pred, Os, Is_x),\ Y = lab(Pred, Os, Is_y),\ Is_y \subset Is_x,$$
$$N = (Label, SizAssign),\ SizAssign = (CA_1, ..., CA_n), and$$
$$CA_i = ((a_i^1, E_i^1), ..., (a_i^m, E_i^m))$$

we define $Sub(N, X, Y) = N'$, where:

$$N' = (Label, SizAssign'),\ SizAssign' = (A_1, ..., A_n),\ and$$
$$A_i = ((a_i^1, Sub(E_i^1, X, Y)), ..., (a_i^m, Sub(E_i^m, X, Y)))$$

We extend the previous definition to a set T of transformation nodes:

$Sub(T, X, Y) = T'$,

where T' is the result of substituting X by Y in every node in T. □

Example 5.1 Consider the transformation node N_1 as defined in Example 4.1, then:

$Sub(N_1, lab(append/3, (3), (2)), lab(append/3, (3), ())) = N_s$, where N_s is:

$$(lab(qsort/2, (2), ()),$$
$$\quad (((2, 0)),$$
$$\quad ((2, sd(lab(append/3, (3), ()), 3, 4, ())))))$$

□

Lemma 1 *Let (T, N_e) be a transformation with entry point N_e, such that there is a set S of n nodes in T, $\{N_1, ..., N_n\}$ labeled with the same label L. Let $T_i = (T - S) \cup \{N_i\}$ for all i, $1 \leq i \leq n$, and EP_i be a transformation node defined as follows:*

$$EP_i = \begin{cases} N_i & \text{if } N_e \in S - \{N_i\} \\ N_e & \text{otherwise.} \end{cases}$$

then for all i, $1 \leq i \leq n$, (T_i, EP_i) is a transformation.

Proof Trivial. □

Lemma 2 *Let (T, N_x) be a transformation whose entry point is the node N_x. Let $N_y \in T$ and $N_z \in T$ be two transformation nodes labeled with Y and Z respectively, such that $Y < Z$. Let N_w be a new entry point defined as:*

$$N_w = \begin{cases} Sub(N_y, Z, Y) & \text{if } N_x = N_z \\ \\ Sub(N_x, Z, Y) & \text{otherwise.} \end{cases}$$

Let $T' = Sub(T, Z, Y)$, *and* $T'' = CN(N_w, T') \cup \{N_w\}$

Then (T', N_w) *and* (T'', N_w) *are two transformation, and* $N_z \notin T''$

Proof Trivial. □

Theorem 2 *If there is a transformation* T *for a size computation specification* X, *then there is an irreducible transformation* T' *for* X.□

Proof It is possible to obtain a finite set of irreducible transformations from T applying the following two processes:

1. Eliminate nodes with the same label (lemma 1). We can obtain a set of transformations: $\{T_1, ..., T_m\}$, which do not contain nodes with the same label.
2. Eliminate nodes that have a label which is less than another label in the same transformation. We can choose any of these T_i and apply successive substitutions to it (lemma 2), obtaining in each step a Transformation T_i^j, defined as T'' in lemma 2. We apply substitutions until we obtain an transformation T_i^K which does not contains pairs of transformation nodes labeled with labels such that one is less than the other one. □

□

Theorem 2 means that we only need to find irreducible transformations to determine whether a procedure is transformable to compute sizes. Obviously, irreducible transformations will result in transformed procedures with potentially less overhead at run-time than the transformations they have been obtained from, but now the problem is to decide which irreducible transformation will have less overhead, or, in other words, which of them will be optimal. The problem of finding such optimal irreducible transformations lies in the fact that we need to use two parameters (number of transformation nodes and number of arguments needed) in the comparison and some transformations may be incomparable, in the sense that one is smaller than the other one on one criteria but the converse is true on the other criteria. In practice we can always assign costs or weights to both argument passing steps and arithmetic operations so that for each transformation we can obtain a function which gives its cost or overhead as a function of the input data sizes. In this case we can compare the cost of irreducible transformations and decide which of them is optimal. In the same way, we can compare the cost of irreducible transformations with the cost of performing the standard size computation, i.e. the one using predefined predicates such as length/2, in order to see how convenient performing the transformation to compute sizes is.

6 Searching for Irreducible Transformations

Since the number of transformation nodes for a given size computation specification is finite, a possible algorithm to find transformations may be to simply generate all possible sets of

transformation nodes and test which of them are irreducible transformations. Note that the number of transformation nodes is restricted by the number of size relations that can be inferred by size analysis [5]. For this reason, if the algorithm does not find any transformation it does not mean that there is not a transformation, but that it is impossible to find a transformation with the inferred information by size analysis. However, some other more efficient approaches are possible.

One possible approach is to follow a top-down algorithm. This approach is based on the generation of *AND-OR* trees, where labels are the OR nodes and transformation nodes are the AND nodes. The search process is similar to SLD-Resolution. In this analogy, we can regard the resolvent in our *SLD-Like* algorithm, as the set of labels for which it is necessary to find transformation nodes labeled with them. Our current substitution, which we call *current transformation*, is the set of transformation nodes assigned to labels, and it will constitute the *answer transformation*. Thus, when the resolvent is empty the current transformation is the answer transformation, which will be irreducible. The entry point of an answer transformation is the transformation node assigned to the label that constitutes the root of the search tree. We represent the current transformation as a list of transformation nodes. Since there may be several labels for a given size computation specification, it is necessary to generate several search trees, with each label being the root of a tree. The search process starts with the resolvent being a label , which is the root of the tree, and an empty current transformation. A resolution step consists of removing a label from the resolvent and assigning to it a transformation node which is labeled with this label and it does not contain labels in its size expressions that are greater than some label of the nodes in the current transformation. This transformation node will be added to the current transformation. After, this the resolvent is modified by adding all the labels that appear in the selected transformation node such that a) they are not yet in the resolvent and b) no identical transformation node appears in the current transformation labeled with such labels.

The transformation nodes are generated with the condition that they do not contain labels in its size expressions that are greater than the label of the node. Once we get all the answer irreducible transformations of all the possible *AND-OR trees* the remaining problem is determining which of them will have the least overhead in the size computation process.

The efficiency of the previous top-down algorithm can be improved if the alternatives for the *OR nodes* are generated with some knowledge regarding which labels will fail. If the base cases of recursive predicates are examined, it is possible to ensure that some labels will fail, and prune the search trees considerably – i.e. to apply a top-down driven bottom-up algorithm. We have built a prototype implementation in Prolog along these lines which makes use of the built-in search capabilities of Prolog to perform such a top-down search.

Another alternative is to apply directly a bottom-up algorithm. In this approach, first transformation nodes are found for the leaves in the call-graph and this information is propagated to find transformation nodes for the ancestors, until we get to the root. Finding a transformation node will imply in this approach the computation of a *fixed-point*.

Example 6.1 Consider the predicate `qsort`/2 as defined in Example 4.1, and suppose we want to transform it to compute the length of its second argument, that is, we want to find a transformation for the size computation specification $(qsort/2, (2))$. We can apply a top-down algorithm. To do this we need to generate some transformation nodes. Consider for example N_1, N_2 and

N_3, where:

N_1 is:

$$(lab(qsort/2,(2),()),\\ \quad(((2,0)),\\ \quad((2,sd(lab(append/3,(3),()),3,4,())))))$$

N_2 is:

$$(lab(qsort/2,(2),()),\\ \quad(((2,0)),\\ \quad((2,sd(lab(append/3,(3),(2)),3,4,\\ \qquad(1+sd(lab(qsort/2,(2),()),2,3,()))))))))$$

and N_3 is:

$$(lab(append/3,(3),(2)),\\ \quad(((3,h(2))),\\ \quad((3,1+sd(lab(append/3,(3),(2)),3,1,(h(2)))))))$$

We can generate a tree for each possible label for the size computation specification $(qsort/2,(2))$, but in this example we are going to generate one for $lab(qsort/2,(2),())$. Thus, the first step is to initialize the resolvent with this label obtaining the initial state:

Resolvent: $[lab(qsort/2,(2),())]$, and Current Transformation: []

Then we remove this label from the resolvent to find a transformation node labeled with it. We first choose N_1 and check that there is no label L appearing the size expressions of N_1 such that there is a node in the current transformation labeled with L and that L is not in the resolvent. The we add N_1 it to the *current transformation list*. After this, the resolvent is modified by adding $lab(append/3,(3),())$ to it. The label $lab(qsort/2,(2),())$ is not added to it because there is still a node in the current transformation labeled with it. At this point the current state is:

Resolvent: $[lab(append/3,(3),())]$, and Current Transformation: $[N_1]$

Then we remove the label from the resolvent and try to find a transformation node labeled with it. But because there is no such node, failure occurs and backtracking is performed, so that the new state is:

Resolvent: $[lab(qsort/2,(2),())]$, and Current Transformation: []

We proceed and find another alternative for $lab(qsort/2,(2),())$, which is N_2, reaching the state:

Resolvent: $[lab(append/3,(3),(2))]$, and Current Transformation: $[N_2]$

The next step is to find a node labeled with $lab(append/3,(3),(2))$. This node is N_3. At this point the resolvent is empty, and the current transformation is an irreducible transformation. The final state is:

Resolvent: [], and Current Transformation: $[N_2, N_3]$

This search may continue until all possible transformations with entry point labeled with $lab(qsort/2,(2),())$ are found. Moreover, all possible search trees can be generated from the possible labels referred to the size computation specification $(qsort/2,(2))$ as its root. □

7 Experimental Results and Advantages of the Method

We have run a series of experiments by using SICStus PROLOG running on a SUN IPC work station to measure the speedup obtained with our predicate transformation technique with respect to which we call "standard approach" to computing term sizes, that is by introducing new calls to predicates that explicitly compute them. For example, by using the Prolog *length*/2 built-in to compute lengths of lists or use other similar built-ins. Although this speedup can be arbitrarily large, we have chosen some benchmarks tailored at allowing us to get some feeling for the performance gain which can be obtained in practice in a number of cases. Table 1 shows execution times for the experiment done with these benchmarks. $\mathbf{T_0}$ is the execution time without size computation. $\mathbf{T_1}$ is the execution time of the size computation with the standard approach. $\mathbf{T_2}$ is the execution time of the predicate transformation approach. $\mathbf{T_1} - \mathbf{T_0}$ and $\mathbf{T_2} - \mathbf{T_0}$ are the overheads due to size computation with the standard and predicate transformation approach respectively. The last column shows the speedup achieved by the predicate transformation approach with respect to the standard one. This speedup is computed according to the following expression:

$$speedup = \frac{(\mathbf{T_1} - \mathbf{T_0}) - (\mathbf{T_2} - \mathbf{T_0})}{\mathbf{T_1} - \mathbf{T_0}} 100 \qquad (1)$$

The first benchmark that we have chosen is the predicate `c`/2, which represents the standard case of a simple list traversal:

```
c([],[]).
c([X|Y],[X|Y1]) :- c(Y,Y1).
```

We assume that we call `c`/2 with the first argument ground and the second one a free variable, and that we need to know the length of the second argument once the goal `c(X, Y)` succeeds. We perform the following transformation, which uses accumulating parameters, so that tail recursion is preserved:

```
trc([],[],S,S).
trc([X|Y],[X|Y1],S1,S) :- S2 is S1 + 1, trc(Y,Y1,S2,S).
```

To measure the execution time with the standard approach we use the following definition of `length`/3, using the built-in `is`/2, so that the execution time is comparable with that of `c`/2, which uses the same built-in:

```
length([], I, I).
length([_|L], I0, I) :- I1 is I0+1, length(L, I1, I).
```

Note that although we could use a more efficient definition of `length`/3, by using more efficient built-ins than `is`/2 (or using a built-in version), we can always do the same for the predicate transformation and use special arithmetic built-ins (as we have done in our implementation). In this case $\mathbf{T_0}$ is the execution time of the goal `c(X, Y)`, $\mathbf{T_1}$ is the execution time of the goal `c(X, Y), length(Y, 0, L)` and $\mathbf{T_2}$ corresponds to the goal `trc(X, Y, 0, L)`.

The second benchmark is the predicate `qsort`/2, in which the lengths of the two output lists of `partition`/4 are computed [3].

The third benchmark is the predicate `q`/2 defined as follows:

```
q([], []).
q([X|Y], [X,X|Y1]) :- X > 7, !, q(Y, Y1).
q([X|Y], [X,X,X|Y1]) :- X =< 7, q(Y, Y1).
```

In this case $\mathbf{T_0}$ is the execution time of the goal `q(X, Y)`, $\mathbf{T_1}$ is the execution time of the goal `q(X, Y), length(Y, 0, L)` and $\mathbf{T_2}$ corresponds to the goal `trq(X, Y, 0, L)`, where `trq`/2 is defined as follows:

```
trq([], [], S, S).
trq([X|Y], [X,X|Y1], S1, S) :- X > 7, !, S2 is S1 + 2, trq(Y, Y1, S2, S).
trq([X|Y], [X,X,X|Y1], S1, S) :- X =< 7, S2 is S1 + 3, trq(Y, Y1, S2, S).
```

Execution times have been measured for different lengths of the input list for these three benchmarks, and the observed speedup is approximately constant in each case.

Finally, the fourth benchmark is the predicate `deriv`/2 (we do not include the corresponding transformation for the sake of brevity).

The observed speed-ups arise from two factors: avoiding additional term traversal and performing less arithmetic operations. As we can see from benchmarks `deriv`/2 and `q`/2, the "standard approach" has to traverse greater data and thus the number of arithmetic operations is greater than in the predicate transformation approach.

Note that another advantage of our approach is that it can take profit of previous size computations so that no recomputation is performed.

Example 7.1 Consider another case – let us assume that we have the goal:

```
q(X), append(Y,X,Z), append(W,X,K)
```

where `X`, `Y` and `W` are ground lists, and `Z` and `K` are unbound variables that will be bound to lists when the goal succeeds. Let us also assume that we are interested in knowing the lengths of `Z` and `K` after the execution of the goal. Using the standard approach we may have:

[3] This size computation is useful to transform predicate `qsort`/2 in order to perform granularity control

```
q(X), append(Y,X,Z), append(W,X,K), length(Z,LZ), length(K,LK)
```

while using the predicate transformation approach we would have:

```
q1o(X,SX), append3o2i(Y,X,Z,SX,SZ), append3o2i(W,X,K,SX,SK)
```

where `q1o(X,SX)` computes the length of `X` (`SX`), which is used by `append3o2i`/5 to compute the lengths of `Z` and `K` (`SZ` and `SK`). In this case the sum of the lengths of the data traversed, which is equivalent to the operations needed to compute the lengths is: $length(X)+length(Y)+length(W)$. In the first case (standard approach) we have: $length(Z)+length(K)$, but since: $length(Z) = length(X)+length(Y)$, and $length(K) = length(X)+length(W)$ we have: $2 * length(X)+length(Y)+length(W)$. □

One might think that a better solution to the first approach would be:

```
q(X), append(Y,X,Z), append(W,X,K), length(X,SX),
length(Y,SY), length(W,SW), SZ is SX + SY, SK is SX + SW
```

but in this case it is necessary to analyze the program to infer that the length of the third argument of `append`/3 is the sum of its two first arguments. This may be easy in some cases, for example for `append`/3, but may be more difficult or impossible in some other cases. This is the case when the length of a list depends not only on the length of other lists but also on its contents. In any case, note that our approach would still take advantage of such optimizations if they can be detected.

However, there are also some cases in which the predicate transformation approach can be more expensive than the standard one. Such cases may appear in connection with backtracking – for example, if `p`/2 were defined differently in such a way that it performed a large amount of backtracking, then it might be better to compute the length of the second argument with the standard approach (using `length`/2). Thus, the predicate transformation approach will perform best when used in deterministic computations. Also, one can construct predicate transformations which perform redundant size computations. Consider, for example, the following definition of a transformation of the qsort/2 predicate, which computes the length of the second argument (output), provided that the length of the first one (input) is supplied at the call:

```
qs2o1i([], [], 0, 0).
qs2o1i([X|I], O, Size_XI, Size_O) :-
     % Size_XI is the length of the input list
     partition3o(X, I, Lg, Sm, Size_Lg),
     Size_Sm is Size_XI  - 1 - Size_Lg,
     qs2o1i(Lg, OLg, Size_Lg, Size_OLg),
     qs2o1i(Sm, OSm, Size_Sm, Size_OSm),
     Size_O is Size_OLg + Size_OSm + 1,
     append(OLg, [X|OSm], O).
```

In this case more arithmetic operations are performed than with the standard approach. But, as pointed in Section 5, this situation can be avoided by obtaining a function which gives the cost

benchmark	T_0 (ms)	T_1 (ms)	T_2 (ms)	$T_1 - T_0$	$T_2 - T_0$	speedup
c/2	202.9	405.69	277.99	202.79	75.09	63.0 %
qsort/2	1218	1495	1343.9	277	125.9	55.3 %
q/2	52,59	90.2	61.69	37.61	9.1	76.7 %
deriv/2	119	3349	239	3110	120	92.9 %

Table 1: Execution times for benchmarks.

of the transformation as a function of the input data sizes, so we can compare this cost with the cost of performing the standard size computation and decide which will be more convenient.

8 Conclusions and Future Work

We have described how predicates can be transformed to compute term sizes at run-time and pointed out the advantages of such transformation. We have also shown a top-down algorithm to find irreducible transformations, which we have implemented in its main part. We are planning on finishing this implementation and evaluating its performance in the granularity application described in [5]. This work is oriented to the development of a complete granularity control system, which can be considered the source of inspiration behind the dynamic term size computation technique presented. In this sense we are working on the integration of this system into a series of other program analysis an transformation tools, that we have implemented, in order to develop improved automatic parallelizing compilers for logic programs.

9 Acknowledgements

We would like to thank Saumya Debray and all the members of the CLIP group (Computational Logic Implementation and Parallelism) at the Facultad de Informática (UPM), F. Ballesteros, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda and L.M. Gómez Henríquez for useful comments during internal presentations of this work. Thanks also to F. Ballesteros and M. Carro for implementing efficient built-ins to replace predefined predicate is/2 in the counting process associated with the dynamic term size computation. The work presented in this paper is supported in part by ESPRIT project 6707 "PARFORCE" and CICYT proyect number TIC93-0976-CE. Pedro López García is also supported in part by grant number AP91-27525988 from the Spanish Department of Education.

References

1. B. Bjerner and S. Holmstrom. A compositional approach to time analysis of first order lazy functional programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.

2. M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

3. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

4. S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

5. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

6. T. Hickey and J. Cohen. Automating Program Analysis. *Journal of ACM*, 35(1), Jan 1988.

7. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

8. D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

9. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

10. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

11. M. Rosendhal. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. MIT Press, 1989.

12. P. Wadler. Strictness analysis aids time analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1988.

13. B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), Sep 1975.

14. X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. A new method for compile time granularity analysis. In *ILPS'91 Workshop on Compilation of Symbolic Languages for Parallel Computers*, 1991.